Diagnosing Incompatibilities in Web service Interactions for Automatic Generation of Adapters

Yehia Taher, Ali Ait-Bachir, Marie-Christine Fauvet, Djamal Benslimane

Diagnosing Incompatibilities in Web service Interactions for Automatic Generation of Adapters

Y. Taher[1,2], A. Aït-Bachir[1], M.-C. Fauvet[1], and D. Benslimane[2]

[1]University of Grenoble LIG Laboratory, 385 rue de la bibliotheque, 38041 Grenoble, France

firstname.lastname@imag.fr

[2]Claude Bernard University of Lyon, LIRIS Laboratory, 43 Bd 11 nov.1918, 69622 Villeurbanne, France

Djamal.benslimane@liris.cnrs.fr

Abstract

Interactions between two applications encapsulated into Web services consist in series of message exchanges that must conform to service interfaces. The study reported in this text aims at dealing with the issues that arise when interactions between two services (a client and a provider) fail because their interfaces are incompatible. This may happen because the provider has evolved and its interface has been modified. It may also happen because the client decided to change for another provider which addresses the same needs but offers a different interface. The contribution of the proposed approach[1] is twofold. First, given two services, all incompatibilities that may exist between their interfaces are automatically detected and classified into patterns. Second, according to the patterns that have been recognised, an adapter is then automatically generated. This latter is intended to act as an intermediate between the client and the provider, therefore seamlessly reconciling interactions between them.

1 Introduction

Web service interactions are composed of message exchanges that trigger specific reactions from the underlying Web services. Accordingly, it is usual to describe a Web service in terms of the message exchanges it engages in, and the actions it performs in response to incoming messages. Service interfaces can be described both from a structural viewpoint (i.e. types of exchanged messages) and from a behavioural viewpoint (i.e. the flow of message exchanges). Thus, the interface of a Web service is defined as the set of messages it can receive and send and the inter-dependencies between these messages.

[1]This work is partially funded by the project Web Intelligence granted by the French Rhône-Alpes Region. Part of it was done when Y.Taher was visiting the BPM Group at QUT, Brisbane under the supervision of Marlon Dumas.

As a Web service evolves, its interface is likely to undergo changes. These changes may lead to the situation where the interface provided by a service no longer matches the interfaces that its peers expect from it. This may result in stopping relationships between the service provider and her/his clients. To enable clients to keep accessing the service despite mismatches introduced by changes in its interface, the provider has to supply an adapter or a mediator. On the other hand, incompatibilities may happen because a service, seen as a client, decided to substitute another service it is used to access, with another one which addresses the same needs but offers a different interface. Actually, each time an evolution or a substitution occurs a new mediator has to be implemented. Developing such pieces of software is a costly and tedious task.

In this paper, we propose an approach whose contribution is twofold: First, given two services, all incompatibilities that may exist between their interfaces are automatically detected and classified into patterns. Second, according to the patterns that have been recognised, an adapter is then automatically generated. The proposed process of adaptation inputs two service specifications and produces the adapter intended to act as an intermediate between the client and the provider, therefore seamlessly reconciling interactions between them. The originality of our approach is that of it addresses detection of all incompatibilities between two interfaces as well as their resolution when related work generally focuses on adaptation only (See [11, 12]).

Moreover, we have designed a boolean language to represent *types of incompatibilities* that may exist between two service interfaces (represented as automata). The idea is that for each type of incompatibility, we will perform a synchronous traversal of the two automata to determine whether or not this type of incompatibility exists between them.

This paper is organized as follow. Section 2 introduces the approach principles. Section 3 presents the approach formal model. Section 4 discusses the implementation and design details. Section 5 presents the related work, and finally we

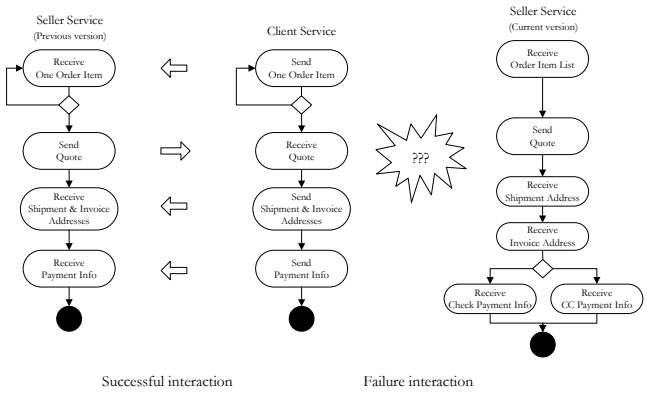

Figure 1. Web service Interactions

conclude in section 6.

2 Background

In this section, we present motivating a scenario, a formal representation of Web service interfaces, as well we introduce language and operators we use farther to model and resolve types of incompatibilities between Web service interfaces.

2.1 Motivation

As an illustrating scenario we consider a service (the seller) which is intended to offer goods for sale. The consumer service is meant to place orders against the seller who in turn, returns a quote, obtains the client's details, the shipment and invoicing addresses, and finally the client proceeds the on-line payment. Figure 1 illustrates the Web service-based process interactions of this example.

The seller service starts by waiting for the client to place an order; to do so, the client has to send as many messages as items to be ordered. To end an order the client sends a specific message. The service is then able to calculate the quote. If the client agrees, it is now required to get the client's details, as well as the shipment and invoicing addresses. The conversation is completed when the payment has been proceeded.

The first part of Figure 1 shows compatible service interfaces of the client and seller services, so interactions can successfully complete between the involved services.

As illustrated by the second part of figure 1, client and seller service interfaces are no more compatible (because, for instance, of an evolution of the seller), so that interactions fail.

The study reported in the following of this paper, aims to automatically detect such incompatibilities between different versions of Web services, and generate the necessary adapters to reconcile these incompatibilities.

2.2 Labelled Transition Systems

Since in our approach we need to reason on Web service interfaces to automatically capturing incompatibilities between them, it is tremendously recommended to model interfaces in a formal representation. That makes it easier to compare interfaces and to capture the existent incompatibilities. For this purpose we chose to model the interface of a Web service using automata *(Labelled Transition Systems (LTS [5])). LTSs* are a fundamental modeling formalism in many areas of computer science where one needs to compare two or more such systems in applications. One of the main reasons for this is that *LTSs* are a well-known paradigm coming up with solid mathematical foundations that can be used to support rigorous analysis and verification. *LTS* consists of a finite set of states and transitions from state to state.

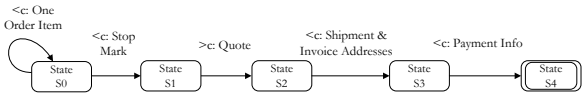

Figure 2. Formalising interfaces with LTSs

In our context, states represent different phases a service may go through during its interaction with clients. Each interaction is considered as a state transition. Transitions are labeled with interaction messages. When a message is sent or received, the corresponding transition is fired. As an example, Figure 2 depicts the LTS modeling the previous interface version of the of the `Seller Service` from the illustrating scenario presented by Figure 1. As expected, the LTS captures the sequence of message interactions the service performs in this scenario. It specifies that the `Seller Service` is initially in the start state s_0 , and that a client begins using the service by sending one or more `(One Order Item)` messages, and the service remains in the same state. Once a message `(Stop Mark)` has been received, the service moves into the state s_1, and so on. Indeed, the LTS depicts the fact that the `Seller Service` is capable of performing the sequence of message interactions : $\langle(<$ c `:One Order Item`$)^*$, ($<$ c `:Stop Mark`), ($>$ c `:Quote`), ($<$ c `:Shipment&Invoice Addresses`), ($<$ c `:Payment Info`)$\rangle$. However, the service cannot perform sequences with prefix $\langle(<$ c `:Order Item List`$)\rangle$ because the start state s_0, does not have any outgoing transitions labelled ($<$ c `:Order Item List`).

2.3 Expression language

In this subsection, we introduce a language including some important concepts and definitions that are used during the incompatibility detection and resolution process. Our language phrases model type of types of *incompatibilities* that may exist between two service interface *LTSs*. The idea is that for each type of incompatibilities, an algorithm that performs a synchronous traversal of the two *LTSs* is run to determine whether or not this incompatibility type exists between them. We made the assumption that each incompatibility

could be captured by exactly one phrase of this language. Further study is already scheduled to assess the validity of this assumption in any situation. In this section, we first introduce this language, then we show how to apply it on the illustrating example given Section 3.1 and eventually we detailed the proposed algorithm.

To build expressions which model incompatibilities, that may exist between two LTSs, we adopt the following notations [4] (examples refer to the LTS depicted in Figure 2):

- $s\bullet$ is the set of outgoing transitions from s.
  (e.g. $S_2\bullet = \{$ <c:Shipment&Invoice Addresses$\}$).
- $\bullet s$ is the set of the incoming transitions for the state s.
  (e.g. $\bullet S_4 = \{$<c:Payment Info$\}$).
- $t\circ$ is the target state of the transition t.
  (e.g. (<c:Shipment&Invoice Addresses)$\circ = S_3$).
- $\circ t$ is the source state of the transition t.
  (e.g. $\circ$(<c:Shipment&Invoice Addresses) = S_2).

The $\circ$ operator (respectively $\bullet$) is generalized to a set of transitions (respectively states). For example, if T is a set of transitions such as: $T = \bigcup_{i=1}^{n}\{t_i\}$ then $T\circ = \bigcup_{i=1}^{n}\{t_i\circ\}$.

Our language supports boolean operators ($\wedge$, $\vee$, $\neq$, etc.) and set operators such as:

- $\parallel s\bullet \parallel$: set cardinality of outgoing transitions of s.
- $s \bullet - s'\bullet$: set difference between outgoing transitions of s and outgoing transitions of s'.

The language also supports filters which build sets of transitions or states satisfying one given property. For instance:

- The operator Polarity(t), applied to a transition t, returns a *string* denoting whatever the interaction message associated to the transition is to be sent (>c:) or received (<c:).
  (e.g Polarity(> *c*:Quote) = {>c:}; that means that the message associated to the transition (> *c*:Quote) is to be *send*.
  Polarity(<c:Payment Info) = {<c:}; that means that the message associated to the transition (<c:Payment Info) is to be *received*.
- The operator Message(t) returns the message (operation argument) to be sent or to be received during a transition t.
  (e.g Message(> *c*:Quote) = {Quote}).
- The operator Type(m), applied to a message m, returns its type structure.
  (e.g Type(Quote) = {T_1}, where:
  $Type\ T_1\langle\ Amount : Float$
  $Currency : String\rangle$).
- The Seq(s) operator, applied to a state *s*, returns a set of transition sequences. Each transition sequence is built starting from one transition $t_i \in s\bullet$.
  (e.g Seq(S_2) = {T}, where: T = { <c:Shipment&Invoice Addresses, <c:Payment Info}).

To check whether or not changes will implie incompatibilities, it is necessary to identify situations when the version P' does not simulate its previous one P. If P' simulates P (denoted $P \preceq P'$) then each interface R required by a client, which is compatible with P (denoted $R \sim P$) remains compatible with P' as shown below. We denote by $\overline{R}$ the opposite interface of R obtained by transforming each message being sent to a message being received and conversely. Given $R \sim P$, thus $\overline{R} \preceq P$ [3]. Due to the transitivity property of the preorder relation of simulation, we have $\overline{R} \preceq P'$ (because $\overline{R} \preceq P$ and $P \preceq P'$). Hence, R conforms to P' ($R \sim P'$). Thus, the detection and the resolution of incompatibilities are relevant only if P' does not simulate P. Section 3.2 below illustrates the language by introducing expressions to detect incompatibility patterns.

3 Incompatibility detection and resolution

In this section, we present our adaptation methodology which inputs two interface automata, detects types of incompatibilities between them, and finally builds (if possible) an automaton adapter. In what follows, we model types of incompatibilities by formal expressions, so that incompatibilities could be automatically detected when comparing two automata modeling different versions of a service interface. To well illustrate the idea, we begin with an illustrating example in Section 3.1, afterward we introduce the principals of the approach in Section 3.2.

3.1 Illustration

Let P be the LTS modeling the interface of a provider service, a client service whose interface is modeled by the *LTS* $\bar{P}$ obtained from P by overturning the polarity of the interaction messages (*i.e., received messages (< c :) becomes sent messages (> c :) and vice versa*) can successfully interact with the provider service. In the sequel of this paper, we denote by P′ the version of the interface resulting from the last modification while P denotes the previous version of that service interface.

Upon such a modification, the client service whose interface is modeled by R (where $R = \bar{P}$) is no more compatible with P′. To deal with such a situation, we need first to detect incompatibilities between P and P′, and then generate an adapter for reconciling at runtime interactions between the Client and Seller services.

As a first step of the incompatibility detection and resolution process, Figure 3 captures some types of incompatibilities detected between P and P′. Namely a One*-to-One incompatibility type is detected at states s_0 and s'_0 of P and P′ respectively. Likely, a One-to-One incompatibility type is detected at states s_1 and s'_1, and so on. A more detailed discussion about each type of these incompatibilities is given in Section 3.2.

In the first step of the detection and resolution process we have focused on comparing the LTSs of the previous and current interface versions of the involved service. Now the objective is to build the LTS modeling the interface of the service adapter. We model an adapter as a service which sets between between two services to be adapted and imitates the required interfaces of both services from each other. That means that the adapter supports the LTS $\bar{P'}$ in its interactions with P′ and the LTS $\bar{R}$ in its interactions with R. For this purpose, we have designed a set of resolution operators each of them is dedicated to handle a specific type of incompatibilities. Such operators, consume message in terms of the requester service interface and produce messages in terms of the provider service interface and vice versa. Figure 4 depicts the sequences of interactions to be performed by the intended adapter. Such a LTS is obtained by composing and wrapping into a map the resolution operators corresponding to types of incompatibilities detected between P and P′.

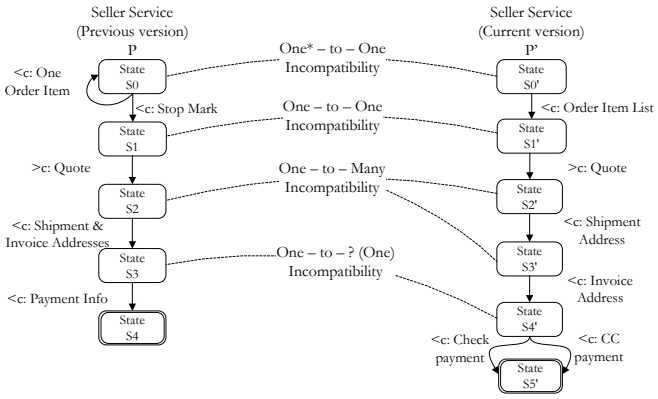

Figure 3. Detection of incompatibility types

Namely, the `Aggregation` resolution operator to reconcile the detected `One*-to-One` incompatibility. The `Matchmake` to deal with the detected `One-to-One` incompatibility, and so on. Section 3.2 details and specifies one by one each of these resolution operators.

3.2 Formalizing incompatibility detection and resolution patterns

To detect incompatibilities, `P` and `P'` are browsed synchronously from their respective initial states s_0 and s'_0. The search seeks out two states `s` and `s'` (belonging respectively to `P` and `P'`) which are such as the sub-automaton starting from `s` in `P` and the one starting from `s'` in `P'` are *incompatible*. The test consists in applying a set of expressions, each of which capturing a specific type of incompatibility.

To each incompatibility type we associate a formal expression and a resolution operator. The formal expression models the incompatibility in terms of states and transitions, so that incompatibilities are likely to be detected when comparing `P` and `P'` automata. However, the resolution operator aims to reconcile the detected incompatibility at execution time. The purpose of this subsection is to sketch this detection and resolution process.

Pattern 1. One*-to-One Incompatibility:

• **Description**: Such a kind of incompatibility arises when an operation loop at a given service interface is replaced by only one operation at a new version. Figure 3 refers to a `One*-to-One` incompatibility detected between `P` and `P'` at states s_0 and s'_0. While the previous version was designed in a way an order of *n* items is realised by consuming *n* (<`c:One Order Item`) messages, the new version expects an order to be realised by consuming only one (<`c:Order Item List`) message.

• **Detection:** During comparing `P` and `P'`, a `One*-to-One` incompatibility is detected if and only if the following expression is evaluated to true:

$\exists\{t_i, t_j\} \subseteq s \bullet \wedge \exists t' \in s' \bullet \mid$
$t_i\circ = s \wedge t_j\circ \neq s \wedge Type(List(Message(t_i))) = Type(Message(t'))$
$Polarity(t') = \{< c :\} \wedge Polarity(t_i) = \{< c :\} \wedge Polarity(t_j) = \{< c :\}$

This expression seeks out two sates s and s' from P and P' respectively, where the target state of one of the outgoing transitions of s is s in itself ($t_i\circ = s$). Therefore, the test follows through comparing whether the type of the message structure obtained by creating a list of messages associated to t_i transition is equivalent to the type of the message associated to t' transition. Finally, the test turns up by verifying that polarity of all messages associated to the of involved transitions is to be received.

• **Resolution:** To deal with `One*-to-One` incompatibility pattern, we have introduced the `Aggregation` resolution operator. This operator is intended to keep consuming and storing messages coming in terms of the previous service interface until receiving a stop mark message indicating that all messages belonging together have been already sent. At this point, the operator is enabled to aggregate stored messages into a single message list represented in terms of the current service interface. Figure 4, illustrates the use of `Aggregation` operator between `R` and `P'`. It starts in a consuming state (s_0, s'_0), in which it is hanging about consuming and storing (<`c:One Order Item`) messages. It is remaining always in the same state unless it receives (<`c:Stop Mark`) message which allows it to move into the next state (s_1, s'_0). At this state, the operator aggregates the set of individual messages into one message containing the order item list, produces the result message to the `Seller Service` and finally ends up into state (s_1, s'_1).

Pattern 2. One-to-One Incompatibility:

• **Description:** When the structure of messages meant to be sent or received in an operation changes, a `One-to-One` incompatibility pattern arises between previous and current versions of that service interface. Referring to our illustrating example, Figure 3 captures `One-to-One` incompatibility between `P` and `P'` at states s_1 and s'_1. While the `Quote` message structure (T_1) consists in two fields, called amount and currency, in the new interface version, such a message structure (T_2) consists in only one field to store both the amount and the currency of the quote.

• **Detection:** a `One-to-One` incompatibility is detected between `P` and `P'` if and only if the following expression is satisfied:

$\exists t \in s \bullet \wedge \exists t' \in s' \bullet \mid$
$t = t' \wedge Type(Message(t)) \neq Type(Message(t')) \wedge Polarity(t) = Polarity(t')$

This expression seeks out two states s and s', from P and P' respectively, whose outgoing transitions t and t' are labeled with the same message ($t = t'$) but the message structure is changed ($Type(Message(t)) \neq Type(Message(t'))$).

• **Resolution:** To deal with such a kind of incompatibility pattern, we have introduced the `Matchmake` resolution operator. Its main task consists in consuming a message of type T_1 and producing a message of type T_2. The message type transformation from T_1 to T_2 is subject to a structural transformation function F_{Match}, provided as a parameter by the adapter designer. Once a message m_1 of type T_1 is consumed by the operator, the function F_{Match} will be applied to in order to produce a message m_2 of type T_2. While a `One-to-One` incompatibility is detected between `P` and `P'` at the states s_1 and s'_1 (Figure 3), we use the `Matchmake` operator to reconcile such a kind of incompatibility between `R` and `P'`. In Figure 4, the `Matchmake` operator starts in a message consuming state (s_1, s'_1). Once a message `Quote` of a structure type T_1 has been consumed (<`c:Quote`), the operator enters

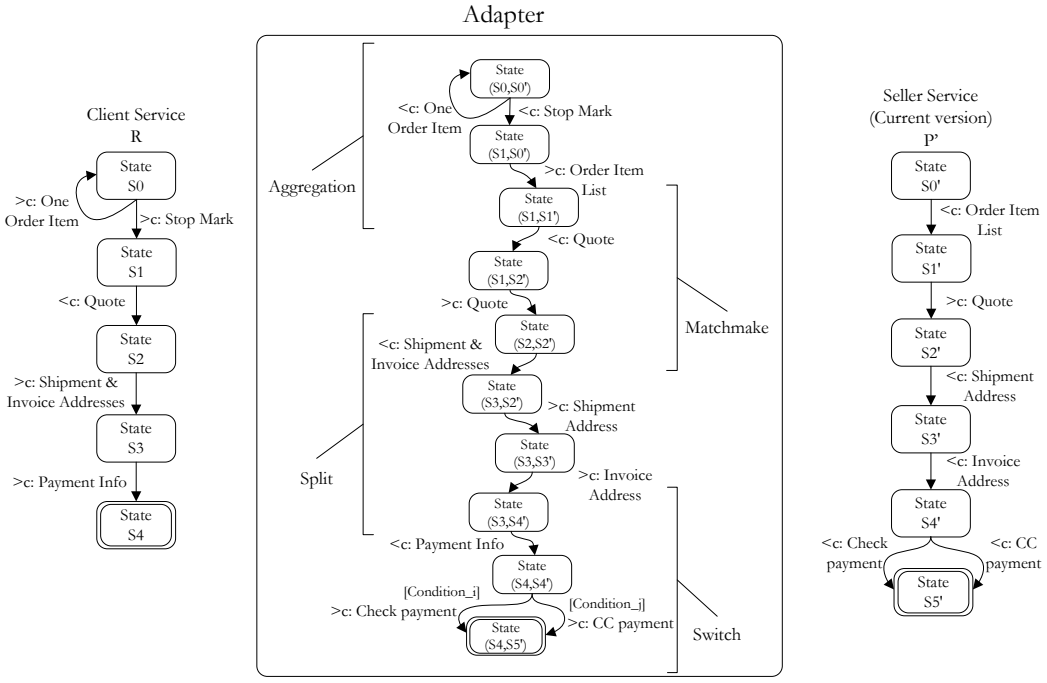

Figure 4. Resolution of detected incompatibilities

into the next state (s_1, s_2'). At this state, it is being able to apply the structural transformation function to the received message and produce a message `Quote` of structure type $\mathtt{T}_2$ (`>c:Quote`). Afterward, the operator enters into its final state (s_2, s_2').

Pattern 3. One-to-Many Incompatibility:

• **Description:** This kind of incompatibility arises when a given operation at a given service interface is changed into a sequence of operations at a new version of that interface. In this change pattern, the message parameter structure of each of the sequence operations is a sub-structure of the message parameter of the mentioned operation at the previous version. Figure 3 illustrates an example of this incompatibility pattern between state $\mathtt{s}_2$ from `P` and states $\mathtt{s}_2'$ and $\mathtt{s}_3'$ from `P'`. As captured by the figure, the operation (`<c:shipment&Invoice Addresses`) is changed into is changed into a sequence of two operations (`<c:Shipment Address`) and (`<c:Invoice Address`) at the current version of that service interface.

• **Detection:** When comparing `P` and `P'`, a `One-to-Many` incompatibility is detected if and only if the following expression is satisfied:

$\exists t \in s \bullet \wedge \exists e \in Seq(s') \mid$
$Type(\bigcup_{i=1}^{\|e\|} Message(e_i)) = Type(Message(t))$
$\wedge Polarity(t) = \{<: c\} \wedge (\forall e_i \in e \mid i \in [1, \| e \|], Polarity(e_i) = \{<: c\})$

The idea is to look for an operation sequence at the current version ($e \in Seq(s')$) such as the union of all message structures of this sequence is the same as one of the previous operation message ($Type(\bigcup_{i=1}^{\|e\|} Message(e_i)) = Type(Message(t))$).

• **Resolution:** To handle such a type of incompatibility pattern, we have introduced the `Split` resolution operator. Such an consists in providing `One-to-Many` structural message transformations. Once a message `m` of type structure `T` has been consumed, a set of predefined functions $\mathtt{F}_{Split_1}$, ..., $\mathtt{F}_{Split_n}$, provided as parameters, are applied to `m` in order to extract and produce `n` sub-messages $\mathtt{m}_1$, ..., $\mathtt{m}_n$ of type structures $\mathtt{T}_1$, ..., $\mathtt{T}_n$ respectively. These sub-messages are considered as input message parameters of the involved sequence operations at the current service interface. Figure 4 illustrates the principal of this solution. In this example, the `Split` operator starts at state (s_2, s_2') by waiting for a message from the `R`. Once, such a message has been received (`<c:Shipment&Invoice Addresses`), the operator stores it and move into the next state (s_3, s_2'). At this state, $\mathtt{F}_{Split_1}$ will be applied to the stored message in order to extract the *shipment address*, trigger the corresponding operation (`>c:Shipment Address`) at the `Seller Service` interface, and enters then into state (s_3, s_3'). Likely, at this state $\mathtt{F}_{Split_2}$ will be applied to the stored message in order to extract the *Invoice Address*, trigger the corresponding operation (`>c:Invoice Address`) at the `Seller Service` and then ends up into state (s_3, s_4').

Pattern 4. One-to-?(One) Incompatibility:

• **Description:** In this change pattern, one operation at a given service interface is changed into two or more operation branches at a new version of that service interface. While at the previous version, the mentioned operation is used to deal with different kind of cases such as achieving payment (`<c:Payment Info`) either by Credit Card or check (Figure 3), the new interface version provides multiple operations each of them is intended to deal with a specific kind of payments. We distinguish between an operation to process credit card payments (`<c: CC Payment`) and another to process check payments (`<c:Check Payment`).

• **Detection:** When comparing P and P′, a One-to-?(One) incompatibility is detected if and only if the following expression is satisfied:
$\exists t \in s \bullet \wedge \exists e \subseteq s' \bullet \wedge \parallel e \parallel \geqslant 2 \mid$
$\forall t' \in e : t' \neq t \wedge Type(Message(t')) = Type(Message(t))$
$\wedge Polarity(t) = \{< c :\} \wedge Polarity(e_i) = \{< c :\}$
This expression looks for a set of two or more operation branches ($\exists e \subseteq s' \bullet \wedge \parallel e \parallel \geqslant 2$), where each branch operation (in the new version) is different from the previous operation ($\forall t' \in e : t' \neq t$). However, the message structure of each branch operation matches the message structure of the previous operation ($Type(t') = Type(t)$).
• **Resolution:** To deal with such a type of incompatibility pattern, we have introduced the Switch resolution operator. This operator is used to map a message issued from a requester service to the right operation at the provider service interface. The behavior of the Switch operator is depicted in Figure 4. It starts in the consuming state (s_3, s'_4), and keeps waiting for receiving the payment message (<c:Payment Info) issued from R. When this happens, the operator moves into the next state (s_4, s'_4). At this state, one of the outgoing transition branches (>c:Check Payment or >c:CC payment) should be fired, and then enters into the final state (s_4, s'_5). Firing one of these transitions depends on the outcome of guard conditions associated to each of those transitions. If the guard associated to the check payment transition is evaluated to true, this transition will be fired and the check payment operation is then triggered at P′. Otherwise, the the other transition will be fired and the credit card operation will be triggered at P′. Indeed, such guard conditions are evaluated by observing dynamically the message content in order to verify what kind of payments it is.

Pattern 5. Many-to-One Incompatibility:
• **Description:** In this change pattern, a sequence of operations at a given service interface are modified into one single operation whose message structure can be computed from the structure union of the messages related to the mentioned sequence of operations. This pattern principal can be considered as the reverse of the One-to-Many pattern principal.
• **Detection:** When comparing P and P′, a Many-to-One incompatibility is detected if and only if the following expression is validated to true:
$\exists e \in Seq(s) \wedge \exists t \in s' \bullet \mid$
$Type(\bigcup_{i=1}^{\parallel e \parallel} Message(e_i)) = Type(Message(t))$
$\wedge Polarity(e_i) = \{> c :\} \wedge Polarity(t) = \{> c :\}$
The expression looks for a sequence of operations at the previous version ($e \in Seq(s)$) such as the union of all message structures of these sequence of operations is the same as one of the new operation message ($Type(\bigcup_{i=1}^{\parallel e \parallel} Message(e_i)) = Type(Message(t'))$).
• **Resolution:** To deal with such a kind of incompatibility pattern, we have introduced the Merge resolution operator. This kind of solution consists in providing Many-to-One structural message transformation. The resolution task consists in consuming n messages of different type T_1, ..., T_n and produce one message of type T. The n-messages merging process is performed by a structural transformation function F_{Merge}, which is provided as parameter by the adapter designer.

Pattern 6. One-to-One* Incompatibility:
• **Description:**

Such a kind of incompatibility arises when an operation at a given service interface is changed into an operation loop at a new version of that interface. This pattern principal can be considered as the reverse of the One*-to-Many pattern principal.
• **Detection:** a One-to-One* incompatibility is detected between P and P′ if and only in the following expression is satisfied:
$\exists t \in s \bullet \wedge \exists \{t'_i, t'_j\} \in s' \bullet \mid$
$t\circ \neq s \wedge t'_i\circ = s' \wedge t'_j\circ \neq S \wedge Type(Message(t)) = Type(List(Message(t'))) \wedge Polarity(t) = \{> c :\} \wedge Polarity(t_i') = \{> c :\}$
This expression seeks out two sates s and s' from P and P' respectively, where the target state of one of the outgoing transitions of s' is s' in itself ($t'_i\circ = s$). Moreover, the test follows through comparing whether the type of the message structure obtained by creating a list of messages associated to t'_i transition is equivalent to the type of the message associated to t transition. Finally, the test ends up by verifying that polarity of all messages associated to the of involved transitions is to be sent.
• **Description:** To handle such a kind of incompatibility pattern, we have introduced the Iteration resolution operator. The main task of this operator consists in receiving a message of multiple elements coming in the same type and structure, and breaking it into a number of sub messages as much as the message element number. Afterward, the obtained set of messages are used then to trigger repeatedly the corresponding operation at the requester service interface.

4 Implementation and design details

As part of our contribution, we provide developers with a framework for performing the automatic incompatibility detection and resolution. The detecttion is processed by a module namely BESERIAL[2] [14] and the resolution runs upon a CEP[3] infrastructure [13]. In what follows, we present the design and implementation details of our tool.

4.1 Algorithm of incompatibility detection

The algorithm implementing the detection phase is detailed in Figure 5. Given P' the new version of the service's interface and P the previous one, the principle of the algorithm is to browse synchronously both LTSs describing P' and P. It proceeds so, starting from s_0 and s'_0 the initial states of P and P' LTSs respectively. The initial call of the algorithm is realized by the expression $Detection(s_0, P, s'_0, P')$.

The algorithm aims at finding all pairs of states s and s' which satisfies one of the incompatibility expressions. It returns a set of triplets $< t, s, s' >$ where t is the incompatibility pattern that has been captured between the sub-automaton of P starting in s and the one of P' starting in s'.

All expressions formalizing incompatibility patterns are first loaded from a database (see line 12). Then, each of which is evaluated considering the current values of the algorithm input data (so-called *current configuration*).

[2] http://www-clips.imag.fr/mrim/User/ali.ait-bachir/webServices/webServices.html
[3] http://www.complexevents.com/

```
1  Detection (si: State; Pi: LTS; sj: State; Pj: LTS ): {Res}
2  { Detection(si,Pi,sj,Pj) is a set of tuples < t, s, s' >
     where t is the incompatibility type between Pi and Pj
     automata (whose initial states are respectively si and
     sj) located at s (one Pi state) and s' (one Pj state). }
3  setRes: {Res} { result variable }
4  P1,P2: LTS ; s1,s2: State ; setRes1: {Res}
5      { intermediary variables }
6  setProg : {coupleStates}
7      { pairs of states to consider next in the progression
       process }
8  IE: {IncompExp} { set of incompatibility types }
9  tEx: IncompExp
10 setRes ← ∅
11 setProg ← ∅
12 IE ← LoadIE() { Loads from a database of expres-
                   sions. }
13 If si• ≠ ∅ then          { condition for no recursive call }
14     For all tEx ∈ IE
15         setRes1 ← Evaluate(tEx,si,Pi,sj,Pj)
16         setRes ← setRes ∪ setRes1
17         If setRes1 ≠ ∅ then
18             { Incompatibilities exist. Which sub-
               automata to consider. }
19         setProg ← setProg ∪ Progression(tEx,si,Pi,sj,Pj)
20     For all c ∈ setProg { Progress to next configura-
                             tions }
21         P1 ← endLTS(Pi,c.s1) ; P2 ← endLTS(Pj,c.s2)
22         setRes ← setRes ∪ Detection(c.s1,P1,c.s2,P2)
23     For all t ∈ si• ∩ sj•  { No incompatibility detected }
24         P1 ← endLTS(Pi,t○) ; P2 ← endLTS(Pj,t○)
25         setRes ← setRes ∪ Detection(t○,P1, t○, P2)
26 Return(setRes)
```

Figure 5. Algorithm of incompatibility detection

Lines 17 to 19 are dedicated to build the input data to be considered by the recursive call. A function named $endLTS(P\text{: }LTS, S\text{: }State)$: LTS is introduced. Given an LTS P and one of its state S, $endLTS(P, S)$ is the fragment (an LTS) of P starting from the state S.

For each detected incompatibility, the pair of LTSs to consider is built before the next recursive call (see lines 20, 20 and 21).

Finally, if no incompatibility is detected, the input data to be considered by the recursive call are target states of the equivalent outgoing transitions, in both LTSs, starting from si and sj (i.e. $si\bullet \cap sj\bullet$) (see lines 23, 24 and 25).

4.2 Framework architecture

Figure 6 depicts our framework architecture, which consists of design and run-time environments. The design-time environment combines incompatibility detection and adapter generation tools. The runtime environment relies on a Complex Event Processing (CEP) infrastructure for messages processing and manipulating.

Starting with the incompatibility detection phase, the `Detection` module hosting and implementing the aforementioned incompatibility detection algorithm, starts in conjunction with the `Compiler` by browsing synchronously the previous and current interface versions of the involved service. For each pairs of states at these interface versions, incompatibility expressions stored in the `Incompatibility Expressions Database`, are evaluated. This process ends up by detecting the set of incompatibilities occurring between the previous and current interface versions of that service.

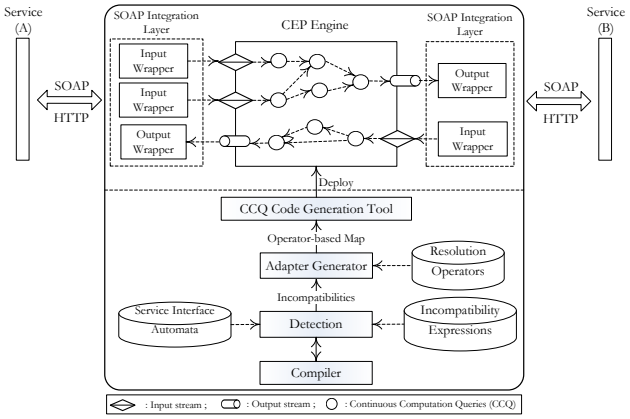

Figure 6. Mediation framework

The `Adapter Generator` module operates upon results exported from the `Detection` module. For each incompatibility element detected between states from the previous and current versions, the `Adapter Generator` identifies the corresponding template resolution operator from the `Resolution Operators Database`. Eventually, the identified operators are then composed together and wrapped into a map. Since this map is a high level of abstraction describing how to adjust and adapt the interactions between the the interacting Web services, it is necessary to translate the template operators based map into an execution-time module and load it to the runtime environment when it comes to deploy the adapter. For this purpose, we have chosen to rely on the insight of `CEP` technology to implement our adaptation model.

`CEP` is a new event driven architecture paradigm that allows of simple events to be aggregated into complex ones. More specifically, `CEP` is a platform for building and running event-based applications that could that could continuously process event streams to detected a specified confluence of events, and trigger a specific action when the events occur. Most of existing `CEP` platforms provide a continuous computation language [2] for specifying queries each of which defining the schema of an event stream to be monitored, defining event patterns to be detected, specifying processing functions and sequencing, and finally declaring output to be generated.

As our adaptation operators consist in consuming incoming messages, processing them, and finally producing adapted messages, we have modeled exchange messages as events, and encoded the message adaptation logic, performed by an operator, in terms of `Continuous Computation Queries` (`CCQ`). This allows a `CEP` infrastructure, combining a `CEP Engine` and `SOAP integration layers`, to acts as a *Web service adapter*. While the `CEP Engine` provides the core service of receiving, correlating, and processing messages against loaded `CCQ`, we have designed

a *SOAP integration layer model* which is mainly intended to assure SOAP interactions between the `CEP Engine` and the involved services. That allows a `CEP Engine` to integrate to *Service Oriented Architecture*. Actually, a *SOAP integration layer* consists of `Input` and `Output Wrappers`. Since the `CCQ` subscribe to *input streams* to consume messages, and publish produced messages into *output streams*, an `Input Wrapper` is used to consume a *SOAP message* issued from the *sender service*, wrap it into a representation appropriate to the `CEP Engine`, and eventually transmit it to the right `Input Stream`. Likely, we associate an `Output Wrapper` to each `Output Stream`. When a request publish a message to a given *output stream*, the `output wrapper` gets such a message and bind it to specific *SOAP plugin* which is intended to trigger the corresponding operation at the current service interface.

5 Related work

The issues tackled in this paper have been partially addressed before, with various points of view. Web service interactions may fail because of interface incompatibilities according to their structural dimension. In this context, reconciling incompatible interactions leads toward transforming message types (using for instance Xpath, XQuery, XSLT). Issues that arise in this context are similar to those widely studied in the data integration area. A mediation-based approach is proposed in [6]. While this approach relies, as ours, on a mediator (called *virtual supplier*) it focuses on structural dimension of interfaces only. An other drawback of this approach is that incompatibilities raised after changes and evolutions of interfaces could not be automatically resolved.

Compatibility test of interfaces has been widely studied in the context of Web service composition. Most of approaches which focus on the behavioral dimension of interfaces rely on equivalence and similarity calculus to check, *at design time*, whether or not interfaces described for instance by automata are compatible (see for example [1, 3, 5]). The behavioral interface describes the structured activities of a business process. Checking interface compatibility is thus based on bi-similarity algorithms [4, 7]. These approaches do not deal with reconciliation issues when incompatibilities occur.

To keep compatibility in a service composition, managing versions is one of the investigated paths in research field. Evolutions and changes are managed by the definition of new versions. In [8], authors propose adapters to reconcile clients, which are compatible with both previous and new versions of the service interface. Nevertheless this proposal focuses on the structural aspects of services only.

Change patterns have been introduced in [9]. These patterns characterize different types of business process evolution. Each pattern models a set of rules which are used by the designer to decide whether or not to propagate changes on executing instances of the modified process or to abort them. As Web services are used as black boxes, this approach does not apply to our context.

6 Conclusion

We presented an approach to automatically detect and resolve incompatibilities that arise in Web services interactions. We studied and formalized incompatibilities that occurs when a provided interface of a Web service evolves and clients and partners can no longer correctly interact with it. Our framework relies on a language to build and to evaluate expressions each of which models an incompatibility pattern (see Section 3.2). These expressions are evaluated in the detection process and the result guides the resolution process. For each incompatibility pattern we associate one adapter pattern. Then we combine these adapter patterns to build one global adapter which is then translated in terms of continuous computation queries and deployed in a *CEP* engine. The adapter proceeds seamlessly and intercept messages between Web service and its environment and make the change completely transparent.
The work presented in this paper opened several research directions. First of all, incompatibility patterns which are not presented in this paper will be investigated and formalized with our language. Some adapter patterns are semi-automatically generated as it require function definitions to model the semantic of the adapter behavior. This step could be performed by adding ontology definitions.

References

[1] Foster, H., Uchitel, S., Magee, J., Kramer, J.: Model-based Verification of Web Service Compositions. In proc. of of ASE 2003, Canada.

[2] Babu, S., Widom, J.: Continuous queries over data streams. ACM SIGMOD Record, v.30 n.3, 2001, 109-120.

[3] Bordeaux, L., Salan, G., Berardi, D., Mecella, M.: When are Two Web Services Compatible? In proc. of TES 2004.

[4] Djikman, R., Dumas, M.: Service-Oriented Design: A Multi-Viewpoint Approach. Journal of Cooperative Information Systems **13**, n**4** 2004, 337–368.

[5] Haddad, S., Melliti, T., Moreaux, P., Rampacek, S.: Modelling Web Services Interoperability. In proc. of ICEIS, 2004.

[6] Altenhofen, M., Boerger, E., Lemcke, J.: An Execution Semantics for Mediation Patterns. In proc. of the BPM'2005.

[7] Martens, A., Moser, S., Gerhardt, A., Funk, K.: Analyzing Compatibility of BPEL Processes. In proc. of AICT-ICIW, 2006, USA.

[8] Kaminski, P., Mller, H., Litoiu, M.: A design technique for evolving web services. In proc. of CASCON, 2006, USA.

[9] Weber, B., Rinderle, S., Reichert, M.: Change Patterns and Change Support Features in Process-Aware Information Systems. In proc. of CAiSE'07.

[10] Benatallah, B., Casati, F., Grigori, D., Nezhad, H. R. M., Toumani, F.: Developing adapters for web services integration. In proc. of CAiSE 2005.

[11] Dumas, M., Spork, M., Wang, K.: Adapt or Perish: Algebra and visual notation for service interface adaptation. In proc. of BPM'2006.

[12] Motahari Nezhad, H. R., Benaltallah, B., Martens, A., Curbera, F., Castai, F.: Semi-Automated Adaptation of Service Interactions. In WWW 2007 Web Services Track, 2007.

[13] Taher, Y., Fauvet, M.-C., Dumas, M., Benslimane, D.: Using CEP Technology to Adapt Messages Exchanged by Web Services. In proc. of WWW 2008, China.

[14] Ait-Bachir, A., Dumas, M.,Fauvet, M.-C.: BESERIAL: Behavioural Service Analyser. In proc. of BPM 2008, Italy.